

# Pseudo Random Number Generators (PRNGs)

By Mohan Atreya (matreya@rsasecurity.com)

## Summary

This is the fourth article in this series. This article aims to educate developers using cryptography on the importance of using good pseudo random number generators (PRNGs) in their applications. The generation of random numbers for use in cryptography is typically not given much importance by developers. This could be due to the fact that developers are not educated to the fact that the strong cryptography is closely linked to strong random numbers. Emphasis is also given to the importance of seeding PRNGs with strong random numbers.

## Introduction

This tutorial assumes that the reader is familiar with basic terms in cryptography such as Public Key cryptography, Secret Key cryptography and Message Digest algorithms. Please review these concepts in Articles 1, 2 & 3 of this series before proceeding further with this tutorial. Alternatively, follow the link (<http://www.rsasecurity.com/rsalabs/faq/index.html>) for a set of frequently asked questions (FAQ's) on e-security from RSA Laboratories.

Cryptographically strong random number generators are a critical component of any good application that uses cryptography. Usually much thought is given to the encryption algorithms and security protocols and almost none to the selection of cryptographically good random number generators. The strength of the encryption keys depends very much on the random numbers. Ciphers such as DES, RC4, RC5 etc. all require a randomly selected encryption/decryption key.

A security protocol is considered secure based on the assumption that the attackers do not have knowledge of the decryption key and cannot determine the decryption key easily by other procedures. Attackers and hackers usually prefer to attack the random number generators rather than the computationally expensive attack on the cryptosystem itself. Once the attacker determines the range of possible key values (commonly referred to as the **Key Space**), the protocol can be broken with a much lesser effort.

True random numbers are totally non-deterministic. They can be obtained from unpredictable phenomena such as electrical and thermal noise from semiconductors, radioactive decay, etc. Access to these sources is next to impossible for software applications. Computers are logical and deterministic by nature and thus are not considered a good source for true random numbers. Software based random number generators can generate pseudo-random numbers at the best and are therefore aptly called Pseudo Random Number Generators (PRNGs).

### Characteristics of good PRNGs

S. No.	Characteristics
1.	Will be random enough to hide <b>patterns</b> and <b>correlations</b> (i.e. distribution of 1's and 0's will have no noticeable pattern).
2.	Will have a large <b>period</b> (i.e. it will repeat itself only after a large number of bits).
3.	Will generate on average as many 1's as 0's.
4.	Will not produce preferred strings such as "01010101".
5.	Will be a simple algorithm with good performance.
6.	Knowledge of some outputs will not help predict past or future outputs.
7.	The internal state of the PRNG will be sufficiently large and unpredictable to avoid exhaustive searches.

### Inside a PRNG

Most of the good PRNGs utilize message digests as mixing algorithms to secure the **seed pool**. The random seed bytes are first fed to the PRNG. The seed pool is then run through a message digest algorithm to create the "**state**" of the PRNG. The "state" is then run through a message digest again to generate the key. The state is then automatically advanced after the "**key**" is generated. **Reseeding** of the PRNG (i.e. adding more random seed material) is usually performed at regular intervals to transform the "state" of the PRNG. Reseeding is performed to thwart attackers from attacking the PRNGs state.

### Seeding PRNGs

The PRNG needs a source of randomness that can provide it with an initial value (also called as a seed) to work with. This process can be viewed as a "**bootstrapping**" operation for the PRNG. The PRNG will expand eventually this random seed into a longer, random bit-stream. Once the seed is fed into the PRNG, the output becomes pseudo-random. Thus, It is obvious that a bad seed can result in a predictable bit-stream being generated, which can potentially cripple a very well designed security protocol. For public-key algorithms, the aim should be to make the effort for searching the seed as difficult as the core mathematical problem of the algorithm itself.

Well-designed applications maintain a seed pool, which is updated whenever the application senses availability of random data. This makes the seed a **moving target** for attackers and can reduce the possibility of seed attacks. The following tables illustrate the **quantitative** and **qualitative** aspects of good seeds.

Quantity
<ul style="list-style-type: none"> <li>• If we use sufficient quantity of the seed material, it would make it impractical for an attacker to perform an exhaustive attack on all the likely seed values</li> </ul>
<ul style="list-style-type: none"> <li>• If we collect enough seed bytes for keys even before the first key generated, we can be sure that we have enough random bits when we need it. This can be viewed as pro-active collection of seed material.</li> </ul>

Quantity
<ul style="list-style-type: none"> <li>• A good quality seed is always a difficult nut to crack. The quality of the seed is related to the entropy value of its bits. (See sidebar for an explanation on entropy).</li> </ul>
<ul style="list-style-type: none"> <li>• The quality of the seed can vary. So, the practical developer will always compliment a small quantity of good quality seed with a large quantity of average quality seeds.</li> </ul>

### Sources for seeds

The following table lists some sources of seed and their relative entropy levels:

System Unique Sources	Variable Sources	External Random Sources
<ul style="list-style-type: none"><li>- Configuration files.</li><li>- Drive configuration.</li><li>- Environment settings.</li></ul>	<ul style="list-style-type: none"><li>- Contents of the screen.</li><li>- Date/time.</li><li>- Last key pressed.</li><li>- Memory statistics.</li><li>- Network statistics.</li><li>- Process statistics.</li><li>- Log file statistics.</li><li>- Program counter for other processes and threads.</li></ul>	<ul style="list-style-type: none"><li>- Cursor position with time.</li><li>- Keystroke timing.</li><li>- Microphone input.</li><li>- Mouse click timing.</li><li>- Mouse movement.</li><li>- Video input.</li></ul>
<div>Less Entropy<span>←</span><span>→</span>High Entropy</div>		

Special care should be taken to handle certain applications such as server applications might not have access to external random sources of data because they are left unattended for long periods of time and lack access to external random sources. The seed source should be built with enough intelligence to make decisions such as, "***not choosing network statistics***" as a seed source just after the machine has been rebooted because all the counters would have been reset to zero. Capturing mouse movements on ***X-Windows*** based servers for the seed is also not a good idea because this data may be available to anyone listening on the wire.

## Rules of thumb

- For every byte of data collected at random, there is one bit of entropy. So, assume that one random byte will result in 1 seed bit.
- The number of random bytes in the seed should at least equal the number of effective bits in a key.
- The application should make the SEED STATE a moving target (i.e. constantly changing). Thus making brute force attack infeasible for an attacker.

## An Example

Objective: To seed a 64-bit DES key (Symmetric Cipher)

**Step 1:** Collect a seed pool of 56 random bytes (a 64-bit DES key comprises 56 effective bits, the rest being parity bits.)

**Step 2:** Select 56 random bits from this seed pool.

Note that when you select 56 bits from a smaller seed pool, it can render the number of possible starting keys from  $2^{56}$  to something smaller. This is undesirable.

## Attacks on seeds

Some of the possible attacks on the seeds of PRNGs are listed below:

S. No.	Description of the attack
1.	<b>Exhaustive seeding search:</b> This attack depends on the fact that the PRNG can be compromised if the attacker can guess the seed bytes that initialized the generator. This attack can be thwarted if you choose good <b>quality</b> seeds in large <b>quantities</b> .
2.	<b>Exhaustive state search:</b> The attacker would attempt to guess the internal state of the PRNG. Exhaustive search of large spaces make this an impractical alternative.
3.	<b>Chosen seed input:</b> The attacker could force the system to reboot by crashing it. Any seed bytes, which are based on the system counters, can now be predicted.

Many more methods exist. Further discussions on these methods of attack are beyond the scope of this discussion.

## Conclusion

In this article, we discussed the internals of PRNGs and their typical characteristics. We also saw how PRNGs can be seeded. Some rules of thumb were highlighted for developers who use cryptography in their applications. We also saw how important seeds are to PRNGs and as a result to crypto-systems themselves.

In the next installment in this series, we will build on the concepts we have discussed so far. We will learn how Digital Signatures and Digital Envelopes work and illustrate how these are used in the real world. We will see how these technologies are used to secure credit card transactions that can be carried out over the Internet using protocols such as SET.

## References

1. A treatise on PKCS Standards from RSA Security, Inc.  
<http://www.rsasecurity.com/rsalabs/pkcs/>
2. Suggestions for Random Number Generation in Software  
<ftp://ftp.rsasecurity.com/pub/pdfs/bull-1.pdf/>

## About the Author

**Mohan Atreya** is a Technical Consultant with RSA Security. He has advanced degrees from National University of Singapore and Nanyang Technological University.

## Acknowledgements

The author acknowledges the significant contributions by Kimberly Getgen and Teo LayPeng to this article.

## About RSA Security Inc.

RSA Security Inc., The Most Trusted Name in e-Security™, helps organizations build secure, trusted foundations for e-business through its RSA SecurID® two-factor authentication, RSA BSAFE® encryption and RSA Keon® digital certificate management systems. With more than a half billion RSA BSAFE-enabled applications in use worldwide, more than six million RSA SecurID users and almost 20 years of industry experience, RSA Security has the proven leadership and innovative technology to address the changing security needs of e-business and bring trust to the new, online economy. RSA Security can be reached at [www.rsasecurity.com](http://www.rsasecurity.com).

## Side Bar

**Entropy:** Entropy is a measure of uncertainty. For example, the gender of a member of a male soccer team has NO ENTROPY. Everyone in a male soccer team is male and there is no uncertainty associated with this.

The more entropy we have in an event, the more random the event is expected to be. Keys used for encryption are expected to be extremely random in nature and thus have high entropy levels. This is necessary because their strength lies in the fact that they remain un-guessable.

Note that if an algorithm specifies that its key size is 64 bits, it does not always mean that it has 64 bits of entropy in the key.

### A Message to Developers:

The RSA BSAFE (<http://www.rsasecurity.com/products/bsafe/>) family of toolkits provides you with all the components you need to make your applications safe and secure. As a developer, you can save many months of development and testing, thus allowing you to focus on your application development and roll out your application with confidence. The BSAFE family comprises the following toolkits:

<b>Core Functionality</b>	<b>BSAFE Toolkit Details</b>
Core Cryptographic Toolkits	BSAFE Crypto-C & BSAFE Crypto-J
Public Key Infrastructure (PKI) Toolkits	BSAFE Cert-C & BSAFE Cert-J
Protocol Level Toolkits	BSAFE SSL-C & BSAFE SSL-J (SSL protocol for point-point security) BSAFE S/MIME-C (S/MIME Protocol for secure messaging) BSAFE WTLS-C (Wireless Transport Layer Security for WAP)